Climate Data Analysis Tools

Version 3

May, 2001

Program for Climate Model Diagnosis and Intercomparison (PCMDI) Lawrence Livermore National Laboratory, Livermore California 94550

Legal Notice

Table of Contents

CHAPTER 1

Getting Started With CDAT

CDAT is an open-source, Python-based set of tools for scientific data analysis and graphics.

1.1 Introduction

1.1.1 A New Paradigm

The software you are about to use, CDAT (Climate Data Analysis Tools), may be unlike any software you may have used before. Rather than a monolithic interface with a fixed number of predetermined commands, CDAT is a set of components that can be programmed under the direction of the user. The set of components can be extended with additional algorithms and compiled code written by the user or written by other members of the community. This kind of extension does not require the cooperation or consent of the CDAT authors or even the recompilation of any of CDAT itself.

CDAT is based on the popular scripting language Python (http://www.python.org). Python is easy to learn, and most importantly for our purposes, easy to extend with your own compiled code and additional modules written in Python itself. In fact, adding your own C, C++, or Fortran can be done nearly automatically using tools that have been written for the purpose. (See “User-contributed packages” on page35.)

Some of the key components supplied by PCMDI include:

- cdms, a package that enables access to many different data file formats;
- cdutil, a package containing utility functions for climatology and averaging;
- genutil, a package containing statistics functions with special features for climate data;
- vcs, a two-dimensional graphics package.

1.1.2 The CDAT Project at SourceForge

CDAT is hosted at SourceForge, a free service provided by VA Linux, Inc. to the Open Source Community. Using SourceForge enables us to have many services for users:

- A release facility, where users can download binary and source releases and see release notes.
- A bug-tracking facility, where users can submit bugs and track their status, and receive mail when they are fixed.
- A mailing list for discussion of CDAT (cdat-discussion@lists.sf.net).

You can use these facilities without registering at SourceForge, but registration, which is quick, easy, and free, will enable you to participate in the fullest possible way. In particular, it is very helpful to us if you are registered when you submit a bug report.

The **CDAT Home Page** is cdat.sf.net. That page documentation and links to related sites. One link is to the **CDAT Project Page**, sf.net/projects/c/cd/cdat. (No, we aren't stuttering; there are so many projects on SourceForge that they had to organize their web site this way). The Project Page contains the bug-tracking, mail list, and download facilities.

1.2 Setting up CDAT

Prerequisites

CDAT assumes you have three packages already installed:

1. Tcl/Tk -- distributions are available from:
   http://tcl.activestate.com
2. Netcdf -- distributions are available from:
   http://www.unidata.ucar.edu/packages/netcdf/
3. Motif -- A free version, LessTif, is included in our source distribution in directory cdat/Tars. You will need to edit and then execute the make_motif shell script.
   Motif is used only in the vcs package now and we hope to eliminate its use entirely in the next release. The file in cdat/Packages/vcs/setup.py is used when building from source to set the location of the library.
   On Solaris LessTif doesn't work because needs to be compiled with the Sun compiler and we could not get it to compile. Our Solaris has a version in /usr/dt/lib and the setup.py file tries to find it there.

Set your environment variable LD_LIBRARY_PATH to include the directories where these libraries are found. For your convenience we supply a script "cdat" that gets installed into the binary directory as bin/cdat. This script can be used to ensure that the environment is correct before executing python. Just edit it accordingly and then execute cdat instead of python. You may need to make similar changes to the script vcdat that starts the browser.

Using the binary distributions

CDAT is available in binary form for a number of platforms. See the CDAT Project Page for the available downloads. It is also available in source form. With the source form, you pick an installation directory. With the binary form, you simply undo the tarball where desired. The top level of the tarball will have a name like cdat-3.0.0.

Either form of CDAT can be installed anywhere on your computer. Wherever you install it, it is important to do two things:

1. Put the 'bin' directory of the installation in your Unix search path so that executing

   where python

   prints the path to the python in your CDAT installation. For example, if you have untarred a binary into /usr/local/, you want /usr/local/cdat-3.0.0/bin in your path, and the above command should print "/usr/local/cdat-3.0.0/bin/python". Depending on your OS, you may need to issue a "rehash" command after doing this. Typically, you are going to want to fix this permanently in your startup files.
2. Set the environment variable LD_LIBRARY_PATH to include the directories where the netcdf library, the Motif library (libXm.a), tcl, and tk are installed. If you do not have these, you will need to obtain them and install them first. Usually they are placed in /usr/local.

Using the source distributions

To build from source, unzip and untar the distribution into any directory of your own and proceed to follow the instructions in the README file at the top level.

A first test

Execute "python" and a prompt >>> should appear. Enter the commands:

```
>>> import cdms
>>> import vcs
```

If you get back another prompt after each command, CDAT is installed correctly. You can leave python by entering "Control-D".

Troubleshooting

If you get an error message about not being able to load a library such as libnetcdf.so, it means you didn't set LD_LIBRARY_PATH.

If python says it can't find modules such as cdms or vcs, it means you don't have our python at the front of your path. Linux, for example, comes with a python already installed so you might be running the wrong one. We have to give you one of our own to make sure your Python version matches the one with which we built all the CDAT components.

CHAPTER 2

Basic tutorials

CDAT comes with a suite of tutorials to help you learn how to use it. You can get these tutorials and sample data from our website.

2.1 Try the Visual CDAT browser first

Before you get into the tutorials, try the vcdat browser first. Execute the script “vcdat” .

2.2 How to get the tutorials and data

The tutorials are designed to introduce you to the most common operations for climate data analysis. They are available as a Python script from our download area under “Tutorials”. If you want to try executing these examples, do these steps:

1. From the Tutorials area of the download facility at cdat.sf.net, download the data files tarball and unpack it.
2. Download the tutorial files. In each tutorial you will need to edit the line(s) near the top that sets the location of the data files to match the location where the data will be on your system.
3. At some point during the tutorials you may wish to also read “Overview of arrays and variables” on pa ge25 for a discussion of

the difference between several array-like abstractions that our software uses: numerical arrays, masked arrays, transient variables, and file variables.

We will now describe the tutorials.

2.3 *getting_started_tutorial.py*

This tutorial is the first one to study. The tutorial consists of three parts:

Example 0: the basics

- How do I open a file?
- How do I see what variables are in the file?
- How do I read a variable?
- How can I get the metadata for the variable?
- How do I get the shape (i.e length of each dimension) of the variable?
- How do I extract a variable and rewrite it to another file?
- How do I extract data at a specific latitude/longitude from the file?
- I do not want to retain the latitude and longitude axes since they are single points. How do I do that?
- How do I extract a subset (or a region) from the file?
- How can I define a region of interest and reuse it without a lot of typing?
- How do I see what the latitude and longitude values extracted are?
- What if I need say the 2nd axis whose name I do not know?
- How do I get the exact region with the precise bounds and not any spillover into adjoining grid cells?

- How do I get the area averaged NINO3 values?
- How do I see what the start time in the data set is?
- How do I extract the data based on time axis values since I know the first time point in the data?
- How do I extract a time slice with specific time start and end?

Example 1: Dealing with data from other sources

- Reading ASCII data like that written by Fortran
- "Masking" or setting a certain value as "Missing data"
- How to create an Axis
- How to set the name of the axis
- How to set the axis units.
- How to create Uniform Latitude and Longitude axes.
- How to get the Bounds.
- How to create a "variable" with all the metadata from an array.
- What are the options for createVariable?
- How to check the shape of the variable
- How to check the metadata or decorations to the variable and its axes.
- How to plot a variable using VCS
- How to write a variable out in a netcdf file.
- Averaging with weights over specified dimensions.
- Specifying weights.
- Generating weights.

Example 2. Masking out data using a land fraction data file.

This example opens a temperature data file and a corresponding land fraction data file. Use the land fraction data to select land/ ocean areas from the temperature data. After masking out data use the averaging routines to compute the area averages.

2.4 Statistics

statistics_tutorial.py illustrates the use of the tools for calculating statistics such as covariances on climate data.

2.5 Dealing with time

Dealing with time is one of the hardest parts of dealing with climate data files. times_tutorial.py illustrates how to use the facilities in CDAT that make dealing with time less painful. Examples are given of:

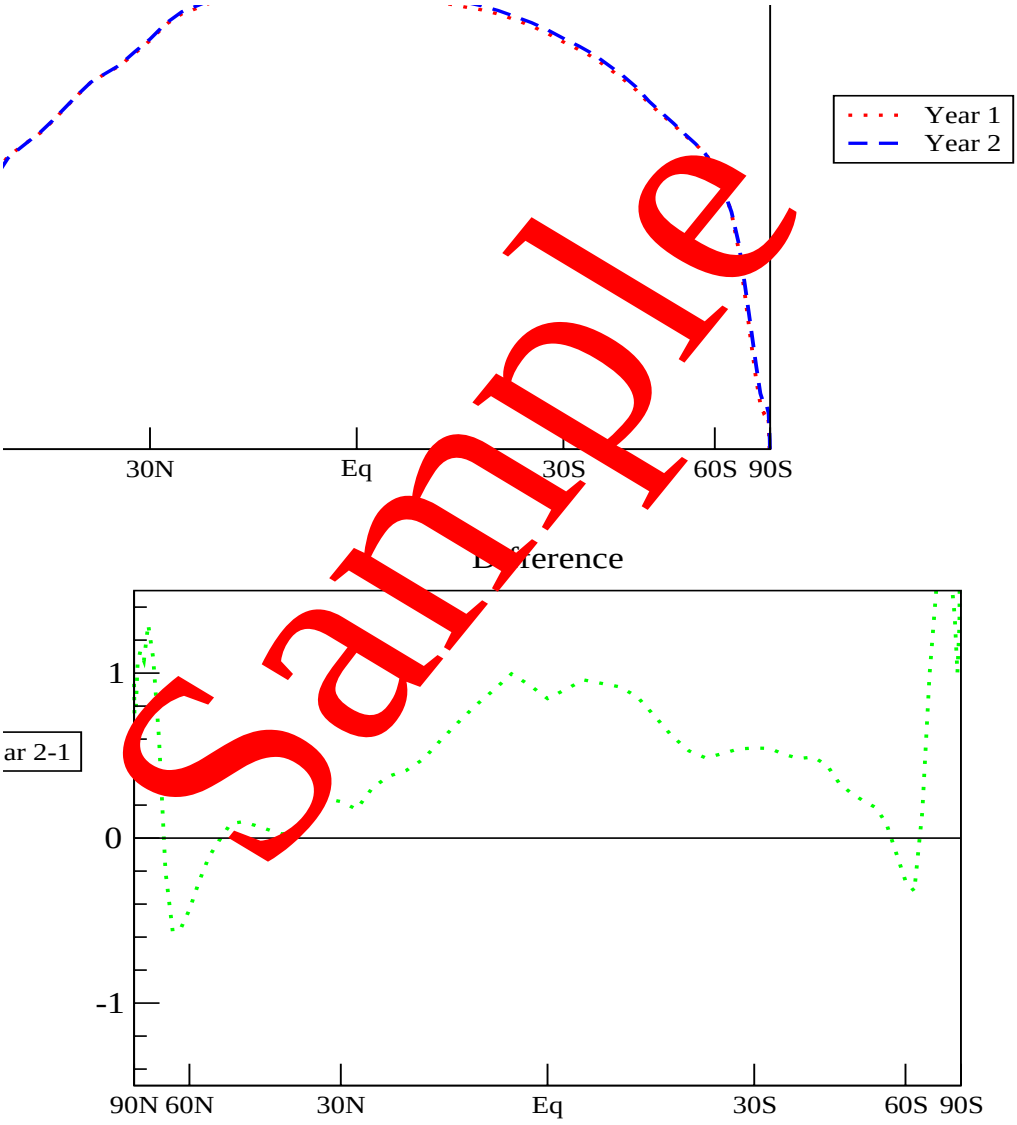
- Compute the climatological DJF
- Calculate departures from that
- Compute the departures from the 1979-1988 period
- Using the predefined seasons:
  JAN, FEB, MAR, APR,...., DEC
  DJF, MAM, JJA, SON
  YEAR -- returns annual means
  ANNUALCYCLE -- returns monthly means for each month of the year
  SEASONALCYCLE -- returns seasonal means for the 4 predefined season
- Compute the annual mean for each year
- Compute the global average for each month

2.6 Plotting with xmgrace

Nothing emphases the fact that CDAT is a collection of tools that can be extended by the user better than the xmgrace module. This module provides an interface to the popular xmgrace utility (which you must have installed yourself separately). One of our users, Charles Doutriaux, who loves xmgrace, built this interface. The xmgrace tutorial will teach you how to use it.

The plot shown on the next page was produced with the xmgrace module tutorial (but it had to be scaled down to fit the page).

A spreadsheet xmgrace.xls is available via the website. This spreadsheet contains detailed information needed by xmgrace users.


Year 1
Year 2
30N
Eq
30S
60S
90S
ar 2-1
1
0
-1
90N
60N
30N
Eq
30S
60S
90S

CHAPTER 3

Learning CDAT

The best way to learn CDAT is to start with the browser, and the the tutorials. This chapter discusses the manuals and online documentation.

3.1 Prefer the Browser and The Tutorials

Don't read this chapter until after you have tried the browser and the tutorials.

3.2 Python Documentation

Python is documented down to the last detail at python.org and the SourceForge website python.sourceforge.net. There are now many books about it. We can recommend the free tutorial available at python.org in the documentation section. The book "Learning Python" from O'Reilly Press is another excellent resource.

CDAT makes heavy use of Numerical Python, a fast array facility for Python. The documentation for Numerical Python is at numpy.sourceforge.net.

Although CDAT itself is not yet available on Windows or MacIntosh, Python is. Python is even available for the Palm Pilot. You can download a

Windows installer and even install Numerical Python on Windows (see numpy.sourceforge.net).

You may also wish to run Python in a nicer environment than a shell. The IDLE environment can be started by entering “idle”. Documentation for IDLE is available via its “help” button. Many people run Python from Emacs. Information about how to configure your editor for writing Python code is available on the Python site.

3.3 CDAT Documentation

The CDAT website cdat.sf.net is your source for documentation, both online and printable.

3.3.1 Using “docstrings”

Python also has a powerful feature: most objects in Python, including the modules, classes, and functions, have documentation strings (“docstrings”) attached to them. The special attribute name “__doc__” (which has two underscores on each end) is used to access this string supplied by the module author. If you have an object x, try the command

```
print x.__doc__
```

3.3.2 Using the happydoc documentation

A utility named “happydoc” is able to generate documentation for modules written in Python. We have used this utility and you will find the resulting documentation on the website. It is very extensive.

3.3.3 Using pydoc to generate documentation

You can also use the pydoc utility. This is a standard facility for extracting documentation from Python installations.

You can generate Python documentation in HTML or text for interactive use.

Interactive use

In the Python interpreter, do "from pydoc import help" to provide online help. Calling help(thing) on a Python object documents the object.

From the shell

At the shell command line outside of Python: Run "pydoc <name>" to show documentation on something. <name> may be the name of a function, module, package, or a dotted reference to a class or function within a module or module in a package. If the argument contains a path segment delimiter (e.g. slash on Unix, backslash on Windows) it is treated as the path to a Python source file. Run "pydoc -k <keyword>" to search for a keyword in the synopsis lines of all available modules.

Starting a browser / server

pydoc -g starts an HTTP server and also pops up a little window for controlling it.

Writing out HTML

Run "pydoc -w <name>" to write out the HTML documentation for a module to a file named "<name>.html".

CHAPTER 4

CDAT Core Modules

CDAT comes with a number of modules. This chapter introduces the core modules, and then gives a conceptual overview of the different kinds of variables and arrays.

4.1 Data acquisition and manipulation

4.1.1 cdms

The principal module used to read, write, and manipulate data is cdms. This module is documented in a large manual available at the CDAT website. Using cdms, you can read and write data, extract portions of data, and compute using that data.

cdms contains a subpackage MV which is generally used for most mathematical operations on cdms-acquired data. For example, to take the square root of a variable, you do:

```
from cdms import MV
y = MV.sqrt(x)
```

This module also contains the basic “selector” mechanism that is used to extract subsets of data. To select by region, such as Northern Hemisphere, see “Statistical utilities” on page 22.

4.1.2 cdms.MV

MV (Masked Variable) is a subpackage of cdms that contains the same mathematical functions found in Numerical Python (Numeric) and the masked array package (MA), specialized to work with cdms variables.

4.1.3 cdtime

Module cdtime contains facilities for specifying times.

4.1.4 regrid

Module regrid helps evaluate a dataset onto a different grid.

4.1.5 cu and pcmdi

Modules cu and pcmdi are obsolete. They are provided for compatibility with CDAT 2.4. Module pcmdi may also contain utilities used only at PCMDI for processing received data sets.

4.2 Graphics

4.2.1 vcs

Graphics are created by creating a vcs Canvas object using the canvas = vcs.init() function call. Then many different kinds of plots can be created by making calls on that canvas, such as canvas.plot (v), where v is a cdms variable.

4.3 Statistical utilities

Packages genutil and cdutil contain utilities for manipulating climate data, such as finding averages, computing climatologies, and extracting data based on time. There are also special selectors defined for easily selecting

data by region, such as the Northern Hemisphere. See the tutorials for lessons on using them.

CHAPTER 5

Overview of arrays and variables

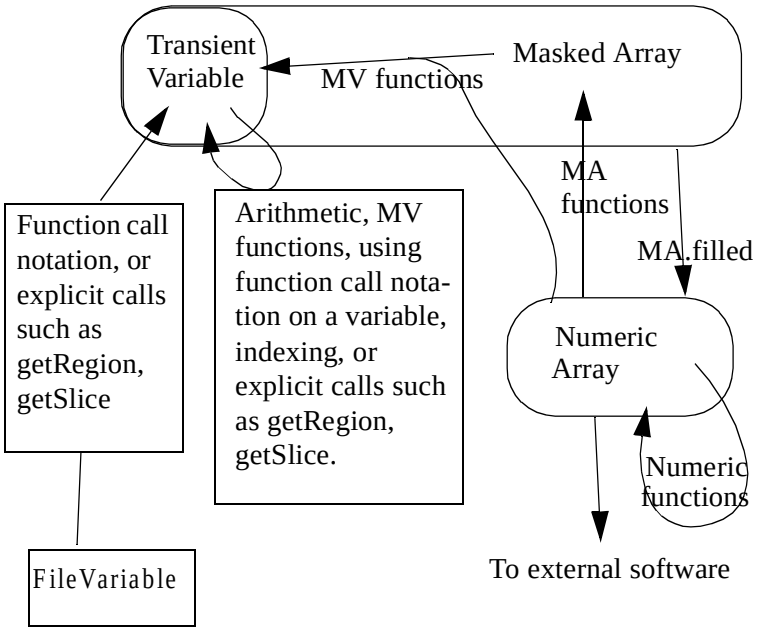
5.1 Why all these layers?

Climate data comes in many different file formats, organized in many different ways. The cdms package gives you a uniform interface so that you can write processing algorithms and graphics that will work with a wide variety of different data formats, not just the kind you first wrote for.

In order to deal honestly with this kind of data, we could have simply invented a “variable” object that contains all the information a climate variable might be associated with. This information includes axis information, informational quantities such as units or where the data came from, etc. However, this approach results in isolation. The other parts of our software, such as graphics, become unusable unless you have a “variable”. If you have calculated some data in some other way, there may be a considerable effort required to “make it into a variable”. Likewise, other Python-based facilities do not know about our “variables” and therefore cannot be easily used from our software, and our software becomes clumsy to use for non-climate data.

Therefore, we opted for interoperability with the rest of the world at the cost of some extra layers of abstraction. In the rest of this section, we will explain these abstractions. Figure 1, “Relationships between array abstractions and functions,” on page26, shows some of the relationships we will discuss.

FIGURE 1. Relationships between array abstractions and functions



5.2 Variables and arrays

Datasets consist of one or more files; these files contain *variables* of various sorts. How the variables are organized within the files becomes irrelevant to a cdms-based program. You might have one file containing all the data, many files each containing one variable, or many files each containing a portion of one or more variables -- but to cdms, the dataset simply is something we can open and variables are arrays that we can get from the dataset. When we do processing on the variables, we produce arrays that exist in memory but not in a file. We call these "*transient variables*" to distinguish them from the permanent kind that are in datasets. If necessary, we

sometimes say "*file variable*" to emphasize that a variable is not transient, although we are playing a little fast and loose with language when we do so, in that such a variable might not be contained in just one file.

Variables have a *domain*; for arrays on an orthogonal grid, the domain consists of a list of *axes*. Conceptually, the i^{th} axis is a one-dimensional array of values the variable v's value at a given set of indices represents the value at the point in space obtained by indexing each axis with its corresponding index value. (Variables that represent quantities on non-orthoganal grids are currently being added to CDAT).

Almost any kind of operation on a file variable, a transient variable, or a combination of the two, will result in a transient variable.

Any variable, transient or not, thus consists of two parts:

- the actual *data* array
- *metadata*, information that helps us interpret what the data means, such as its *domain*, name-value pairs we call *attributes*, and some information about whether which, if any, data locations are *invalid* or *missing* data.

In order to facilitate interoperability with other Python modules, we want to represent the *data* in a variable with an array from the Numeric module, the standard extension to Python used for scientific data.

Numeric arrays have no concept of a missing value; to represent that, we use an array from the MA (masked array) package. If we need to convert a masked array to a Numeric array, MA.filled or the filled method can be used to produce a Numeric array with the missing values replaced by a value we choose, for example, 1.e20.

MA arrays have no concept of a domain or attributes. Since it is both possible and convenient to carry such information through some operations, we have a third layer of abstraction, the variable. In our software, a transient variable (but not a file variable) is an MA too -- we use an object-oriented

concept called inheritance. This means a transient variable can be used in any context in which an MA is expected.

A special submodule, MV, of cdms, is provided to supply the same set of functions that MA and Numeric provide, such as sqrt, transpose, and average. The function in MV return transient variables.

For example, if x is a transient variable, cdms.MV.average(x,axis=1) would average the values of x over its axis that has index 1 (the second one). The resulting MV would have the same domain as x but with the second axis removed.

The benefit of this approach is that, for example, if we average over time, and then plot the result, the plotting routines can still be aware that the other dimensions represent latitude and longitude and draw the continental outlines, do projections correctly, etc. If the result were merely the averaged data, that interpretation of it would have been lost.

5.3 Traversing the hierarchy

In increasing order of complexity, then, we have Numeric arrays as the simplist, MAs next, and then variables, which are available in two flavors, file and transient. What follows is a “cookbook” of how to get from one abstraction to the next and back again. See Figur e1 on page26.

5.3.1 Constructing Numeric arrays

As explained in the Numeric Python manual, a Numeric array can be constructed in many ways. The basic usage is to apply the array “constructor” Numeric.array:

```
x = Numeric.array (s)
```

where s can be a Python list, tuple, or another Numeric array. If you do not require a separate copy of the data, copy=0 can be added. If you want to control the type of the data, you can supply a typecode, usually in the form

of one of the abstract constants supplied in Numeric for this purpose. For example,

```
y = Numeric.array([1,2,3], Numeric.Float)
```

would create y as an array containing the floating-point numbers 1., 2., and 3. By the way, a very frequent beginner error is to say something like:

```
y = Numeric.array(1,2,3)  # Error !
```

which is an error that will result in a very strange message:

```
>>> import Numeric
>>> y = Numeric.array(1,2,3)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in ?
    y = Numeric.array(1,2,3)
TypeError: typecode argument must be a string.
>>>
```

What has happened is that the second argument to Numeric.array was the number 2, and Numeric is expecting one of its typecode letters such as ‘d’ in that position.

5.3.2 Numeric to MA

Given a Numeric array x, if we wish to construct a masked array, it will be one of these cases:

1. We want to treat x as a masked array xm, but none of its values are currently missing, and we are wish to share x’s data space, so that a modification to xm is also a modification to x.

   Solution: xm = MA.masked_array (x) or
   xm = MA.array (x, copy=0)

2. We want to treat x as a masked array xm with no values considered missing and without sharing x’s space.

   Solution: xm = MA.array (x)

3. We want to treat x as a masked array xm with a certain value v treated as a missing value.

   Solution: xm = MA.masked_value (x, v). See the MA manual for other arguments, such as controlling the precision with which a value in x must be equal to v in order to be considered missing.

4. We want to treat x as a masked array xm but with those values considered missing that correspond to non-zero values in an array m of zeros and ones.

   Solution: xm = MA.array(x, mask=m). Again, see the manual for more options on this constructor.

5. The array x is of a numeric type and we want to mask all those values greater than a certain value v.

   Solution: xm = MA.masked_greater (x, v). Similarly, there are functions masked_greater_equal, masked_less, masked_less_equal, masked_equal.

5.3.3 Numeric or MA to Transient Variable

The array constructor cdms.MV.array is similar to the MA.array constructor but allows additional arguments specifying the metadata. For full usage, see the CDMS manual. The options discussed in the previous section will produce transient variables as long as you use the functions in cdms.MV instead of those in MA.

The most frequent additional argument to cdms.MV.array is to specify a list of axes using the axes=alist argument. Often these axes have been extracted from another variable using getAxis or getAxisList, or created from data.

For example, if you were going to massage variable v using some process that returns a Numeric array, and in the process remove its time dimension, you might do something like this:

```
>>> f=cdms.open('/home/dubois/clt.nc')
>>> v=f['clt']
```

```
>>> alist = v.getAxisList(omit='time')
>>> u = massage(w)
>>> x = cdms.MV.array(u, axes=alist)
```

Now, x is again a transient variable and it has the geographic information reattached so that plots will show the continents.

Note that it is usually not necessary to do this. Any ordinary arithmetic, and any use of the functions in MV, will return transient variables with appropriate axes.

CHAPTER 6

Creating New Packages

CDAT is a collaboration. You can be part of the collaboration.You don't need permission. You don't need PCMDI's approval. You don't need PCMDI's programmers to add your algorithms. Just do it.

6.1 Adding your science

One of CDAT's strengths is that it is an open system. You can add your own software written in C, Python, or Fortran. The easiest way to learn to do this is to copy our examples. Get the CDAT source distribution and look for sub-directory 'contrib' in the top-level directory. The README file in contrib explains what to do.

There are tools that may be useful to you.

- The SWIG utility (Simplified Wrapper and Interface Generator, www.swig.org) can wrap C and C++ routines.
- Pyfort (pyfortran.sourceforge.net) connects Fortran routines to Python.

Depending on your needs, you may wish to use a layer of Python along with the automatically created interface, in order to make a nicer interface or to use the Fortran or C simply as computational engines. An example of this is the EOF package described below: it uses a Fortran linear algebra routine to enhance performance, but the "science" is in Python.

If you follow the protocols in ‘contrib’ then your package can be added to the PCMDI distribution as well. Just send it to us and be sure to include a README that explains:

- How to use the package
- Contact information about the author.

You may also be able to generate useful documentation by executing the routines happydoc or pydoc. happydoc works only on Python code; pydoc works on the installed modules. Both routines print help packages if executed with the argument, ‘--help’, and both are already installed in your cdat ‘bin’ directory.

CHAPTER 7

User-contributed packages

Here are descriptions of the packages contributed to the “contrib” section of the CDAT source.

If you have the source distribution, use the README files in the subdirectories of the contrib directory for full documentation. Try running pydoc -w with the name of the package as an argument to create a web page showing the package’s interface.

Due to the very nature of the user-contributed packages, the following list of available packages and their exact capabilities may be incomplete or inaccurate. PCMDI does not maintain all of these packages, and is not responsible for fixing bugs in them -- please contact the author.

7.1 Connecting to Numerical Python packages

Many packages that support Numerical Python arrays have been developed. You can find links to some of them on the CDAT website.

These packages work with Numerical Python arrays. Since the data obtained from files in CDAT can be converted to Numerical Python arrays, you can pass arguments from CDAT to these routines.

The one problem to consider is that of missing data. Numerical Python arrays do not have the concept of missing values. Therefore, you must convert your CDAT quantities to Numerical arrays. An attempt is usually made to do this automatically but it will fail if there are missing values.

To convert an MV (masked array or transient cmds variable) to a Numeric array, you can use the *filled* method or function:

```
from cdms import MV
x = ...some data...
numeric_array = MV.filled (x)
#or
numeric_array = x.filled()
```

The difference is that the first form will work whether x is a Numeric array, list, masked array, or transient variable, while the latter only works for transient variables and masked arrays. Both forms can take an additional argument *fill_value* specifying the value to use to replace any missing values; if you don't give it a default value for that array's type is used.

One useful package, Konrad Hinsen's Scientific Python, is included in your CDAT documentation. The documentation for it is available via a link from the CDAT website.

Package asciidata

Author: Paul Dubois (dubois1@llnl.gov)

Summary: Package asciidata reads data from ASCII text files.

Reads text files written by such programs as spreadsheets, in which data has been written as comma, tab, or space-separated numbers with a header line that names the fields. Using the functions in asciidata, you can convert these columns into Numerical arrays, with control over the type/precision of these arrays.

Example

```
import asciidata
time, pressure = asciidata.comma_separated('myfile.csv')
```

Documentation: pydoc -w asciidata

Scientific Python also contains a subpackage IO that contains other useful facilities of this type. In particular there is a useful package for reading Fortran-like formatted output.

Package binaryio

Author: Paul F. Dubois

Summary: Read and write Fortran unformatted i/o files.

These are the files that you read and write in Fortran with statements like read(7) or write(7). Such files have an unspecified format and are platform and compiler dependent. They are NOT portable. Contrary to popular opinion, they are NOT standard. The standard only specifies their existence and behavior, not the details of their implementation, and since there is no one obvious implementation, Fortran compilers *do* vary. We suggest writing netcdf files instead, using the facilities in cdms.

Documentation: pydoc -w binaryio. A similar package is in Scientific Python.

Usage summary:

```
binaryio: Fortran unformatted io
   Uses Fortran wrapper module "binout"
Usage:
   from binaryio import *
   iunit = bincreate('filename')
   binwrite(iunit, some_array)  (up to 4 dimensions)
   binclose(iunit)
   iunit = binopen('filename')
  y = binread(iunit, n, ...)   (1-4 dimensions)
   binclose(iunit)
```

Note that reads and writes must be paired exactly. Errors will cause a Fortran STOP that cannot be recovered from. You must know (or have written earlier in the file) the sizes of each array.

All data is stored as 32-bit floats.

Package eof

Author: Paul Dubois (dubois1@llnl.gov) based on work by Ken Sperber and Ben Santer.

Summary: Calculates Explicit Orthonormal Functions of either one variable or two variables jointly.

Having selected some data, the key call is to create an instance of eof.Eof giving one or two arguments. In this example, a portion of the variable ‘u’ is analyzed. After the result is returned, it is an object with attributes containing such things as the principal components and the percent of variance explained. Optional arguments are available for controlling the subtraction of the mean from the data, the weighting by latitude, and the number of components to compute.

This routine is computationally efficient, solving the problem in either the normal space or the dual space in order to minimize computations. Nonetheless, it is possible that this routine will require substantial time and space if used on a large amount of data. This cost is determined by the smaller of the number of time points and the total number of space points.

Documentation: pydoc -w eof.Eof

Example

```
import cdms, vcs
from eof import Eof

f=cdms.open('/home/dubois/clt.nc')
u = f(‘u’, latitude=(-20,40), longitude=(60, 120))
result = Eof(u)
principal_components = result.principal_components
print "Percent explained", result.percent_explained
x=vcs.init()
vcs.pauser.pause(3)
print len(principal_components)
```

```
for y in principal_components:
   x.isofill(y)
   x.clear()
u1 = v.subRegion(latitude=(amr[0], amr[1], 'cc'),
            longitude=(amr[2],amr[3],'cc'), order='xyt')
result2 = Eof(u, number_of_components=4,
mean_choice=12)
print "Percent explained", result.percent_explained
```

Package lmoments

Author: Michael Werner

Summary: An interface to an L-moments library by J. R. M. Hosking.

This package is an interface to a Fortran library. The calling sequence from Python differs from the Fortran convention. In general, output and temporary arguments are not supplied in making the Python call, and output arguments are returned as values of the function.

Documentation: pydoc -w lmoments to see list of functions. pydoc -w lmoments.pelexp, or other function name, for the particular. See also documentation for Pyfort at pyfortran.sourceforge.net for further details on argument conventions. If built from source, a file flmoments.txt appears which gives the Python calling sequences.

Package regridpack

Author: Clyde Dease

Summary: Interface to regridpack

Documentation: This package contains a Python interface to the subroutine library regridpack.

pydoc -w adamsregrid Documentation online at cdat.sourceforge.net. See also documentation for Pyfort at pyfortran.sourceforge.net for further details on argument conventions.

Package sphere (spherepack)

Author: Clyde Dease

Summary: Interface to Spherepack

Documentation: This package contains a Python interface to the subroutine library Spherepack.

pydoc -w sphere to see list of functions. Documentation online at cdat.sourceforge.net. See also documentation for Pyfort at pyfortran.sourceforge.net for further details on argument conventions.

Package trends

Author: Pyfort wrapping by Paul Dubois of a routine by Ben Santer

Summary: Computes variance estimate taking auto-correlation into account.

Documentation:

```
import reg_arl from trends
rneff, result, res, cxx, rxx = reg_arl (lag, x, y)
    integer lag         Max lag for autocorrelations.
    real x(n1)          Independent variable
    real y(n1)          Dependent variable
    real, intent(out):: rneff         !Effective sample size
    real, intent(out):: result(31)    !Array of linear regression
results
    real, intent(out):: res(n1)     !Residuals from linear regres-
sion
    real, intent(out):: cxx(1 + lag)   !Autocovariance function
    real, intent(out):: rxx(1 + lag)   !Autocorrelation function
```

Package ort

Author: Curt Covey

Summary: Read data from an Oort file.

Documentation: Module ort contains one Fortran function, read1f:

Calling sequence:

```
import ort
lon, lat, data, nr = ort.read1f(filename, maxsta, nvarbs,
                                nlevels)
```

Input:

```
    character*(*) filename    ! name of the file to be read
! max number of stations (soundings) possible
    integer maxsta
! number of variables and P-levels in each sounding
    integer nvarbs, nlevels
```

Output:

```
 ! longitudes / latitudes of the stations
    real, intent(out):: lon(maxsta), lat(maxsta)
! sounding data
    real , intent(out):: data(nvarbs, nlevels, maxsta)
 ! actual number of stations with data
    integer , intent(out):: nr
```

Package grads

Author: Mike Fiorino

Summary: The grads module supplies an interface to cdms that will be familiar to users of GrADS.

Documentation: See the CDAT website for documentation.

Index

W